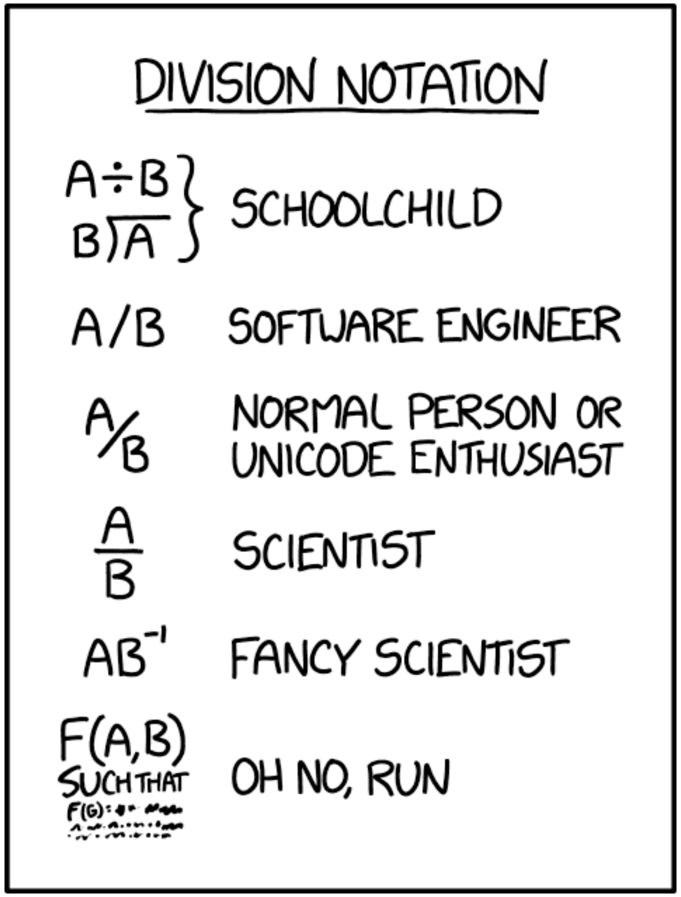
CSE 331 Spring 2025

Specifications



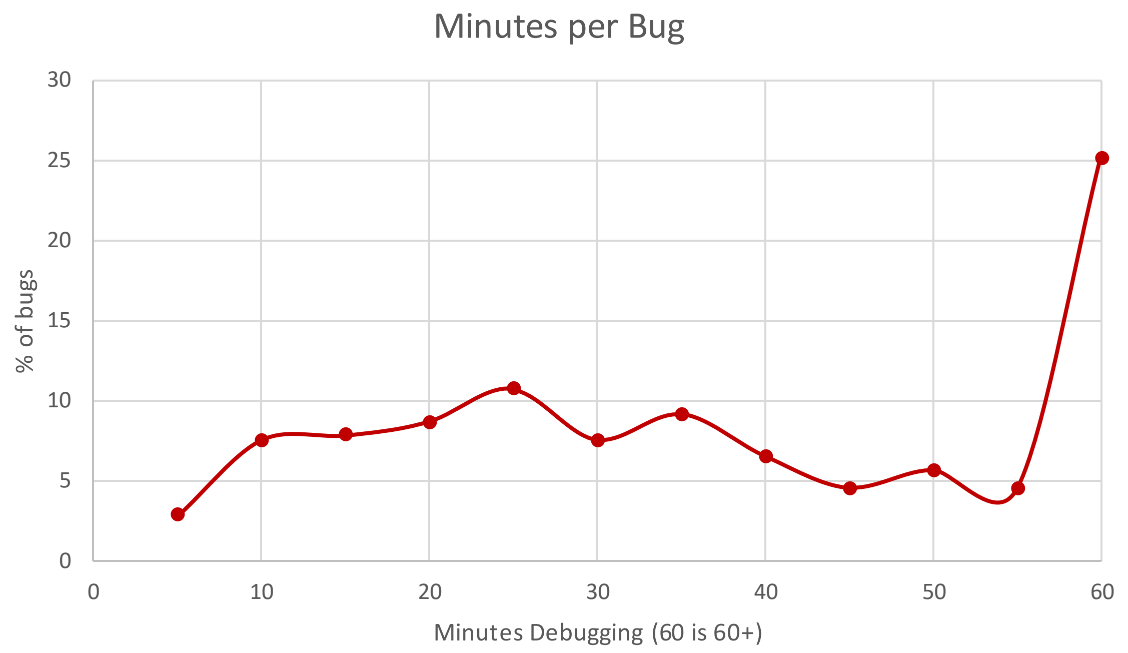
xkcd #2687

Matt Wang

& Ali, Alice, Andrew, Anmol, Antonio, Connor, Edison, Helena, Jonathan, Katherine, Lauren, Lawrence, Mayee, Omar, Riva, Saan, and Yusong

A Brief Look at HW: Time Spent

- Time spent per bug: ~ 42 min/bug, 25% > 1 hour
- ~ 1 bug per 22 lines of code (harder with JSX)



- Long tail is making itself visible…

HW2 & Mutation

- Was the bug due to a disallowed mutation?
  - students reported 'yes' for 11.5% of bugs
  - such bugs took **>40% longer to debug on average**

HW2 Debugging via User Report (1/2)

- **User reports the following bug:**

  "Sometimes, I can't click on one of the markers.
  Usually, it it works fine. But occasionally, you can't click on it."

- **First step is to figure out how to reproduce it**
  - **can't debug otherwise**

    wouldn't know that you've fixed the bug
  - **key reason why event-driven debugging is harder**

    command-line failure is instantly reproducible
  - **debugging a crash is easier than a non-crash!**

    crash comes with a stack trace (line of code with a failure)

HW2 Debugging via User Report (2/2)

- Eventually, you find a way to reproduce it
  - no longer clickable after you move it very far away
- To debug, you must learn how `App.tsx` works
  - markers are stored in some kind of tree
  - searches the tree to find markers near the click
- To debug, you must learn how `marker_tree.ts` works
  - internal tree nodes split into NW, NE, SE, SW regions
  - marker was inserted into the correct region
  - when you search for it, it's no longer in the right region

One "Solution" to HW2 (1/2)

```
type EditorState = {
  newMarker: Marker;
  …
};

doNameChange = (evt: ChangeEvent<…>): void => {
  this.state.newMarker.name = evt.target.value;
  this.setState({newName: evt.target.value});
};

doSaveClick = (evt: MouseEvent<…>): void => {
  this.props.onSaveClick(newMarker.name, …);
};
```

already suspicious…
mutating `this.state` **directly**

One "Solution" to HW2 (2/2)

```
constructor(props) {
  super(props);

  this.state = {newMarker: this.props.marker, …};
}

doMoveToChange = (evt: ChangeEvent<…>): void => {
  const bldg = findBuildingByName(evt.target.value);
  newMarker.location = bldg.location;
  this.setState({moveTo: evt.target.value});
};
```

- **Starting to get nervous…**
  - are we allowed to mutate that marker?
  - **no!** that location is a key in a tree

Staying Safe in 331

1. Do not use mutable state
   - don’t need to think about aliasing at all
   - any number of aliases is fine

2. Do not allow aliases to *mutable* state
   a) do not hand out aliases yourself
   b) make a copy of anything you want to keep

ensures only <u>one</u> reference to the object (no aliases)

- For 331, mutable aliasing across files is a <u>bug</u>!
  - gives other parts the ability to break your code
  - we will stick to these simple strategies for avoiding it

Rules of Thumb: Mutation XOR Aliasing

Client Side

1. Data is small
   - anything on screen is O(1)
2. Aliasing is common
   - UI design forces modules
   - data is widely shared

Rule: avoid mutation
- create new values instead
- performance will be fine
- (local-only mutation can be OK)

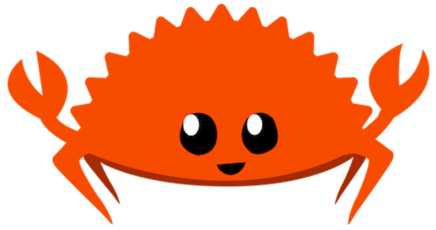
Server Side

1. Data is large
   - efficiency matters
2. Aliasing is avoidable
   - you decide on modules
   - data is not widely shared

Rule: avoid aliases
- do not allow aliases to your data
- hand out copies not aliases
- (good enough for us in 331)

Language Features & Aliasing

- Most recent languages have some answer to this…
- Java chose to make `String` immutable
  - most keys in maps are strings
  - hugely controversial at the time, but great decision
- Python chose to only allow immutable keys in maps
  - only numbers, strings, and tuples allowed
  - surprisingly, not that inconvenient



- Rust has built-in support for "mutation XOR aliasing"
  - ownership of value can be "borrowed" and returned
  - type system ensures there is only one usable alias

Readonly in TypeScript (1/2)

- TypeScript can ensure values aren't modified
  - extremely useful!
  - but, only a compile-time check (not a runtime guarantee)

- Readonly tuples:

```
type IntPair = readonly [bigint, bigint];
```

- Readonly fields of records:

```
type IntPoint = {readonly x: bigint,
                 readonly y: bigint};
```

Readonly in TypeScript (2/2)

- **Readonly fields of records:**

```
type IntPoint = {readonly x: bigint,
                 readonly y: bigint};
```

- **Readonly records:**

```
type IntPoint = Readonly<{x: bigint, y: bigint}>;
```

  – `this.props` is `Readonly<MyPropsType>`

- **More readonly…**

```
ReadonlyArray<bigint>
ReadonlyMap<string, bigint>
ReadonlySet<string>
```

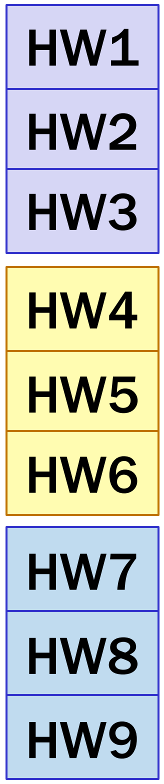
comfy-tslint

- we’ve written a TS linter for this class that enforces some of our conventions, e.g.
  - requiring type annotations for functions
  - disallowing the any type
  - naming & structure conventions for React methods
- available to you…
  - as a VSCode extension
  - as an npm module (that you can run yourself)
- please:
  - take a careful look at the HW3 spec + autograder
  - briefly read the website page on comfy-tslint

Precise Specifications

Where We Are in the Course

Nine assignments split into these groups:



learn to write more complex apps
practice debugging

learn how to be <u>100% sure</u> the code is correct
(most of the work done on *paper*)

Correctness and Specifications

- Correctness requires a <u>definition</u> of the correct answer

- Description must be <u>precise</u>
  - can't have disagreement about what is correct

- Informal descriptions (English) are usually imprecise
  - necessary to "formalize" the English

    turn the English into a precise *mathematical* definition

  - professionals are *very* good at this

    usually just give English definitions

    important skill to practice

  - we will start out completely formal to make it easier

Kinds of Specifications

- **Imperative** specification says <u>how</u> to calculate the answer
  - lays out the exact steps to perform to get the answer
- **Declarative** specification says <u>what</u> the answer looks like
  - does not say how to calculate it
  - up to us to ensure that our code satisfies the spec
- Can implement a *different* imperative specification
  - again, up to us to ensure that our code satisfies the spec

Example: Imperative Specification (abs)

- Absolute value: $|x| = x$ if $x \geq 0$ and $-x$ otherwise
  - definition is an "if" statement

```
const abs = (x: bigint): bigint => {
  if (x >= 0n) {
    return x;
  } else {
    return -x;
  }
}
```

just translating math to TypeScript

Example: Declarative Specification (sub)

- <u>**Subtraction**</u> $(a - b)$: **return** x **such that** $b + x = a$
  - **can see that** $b + (a - b) = b + a - b = a$

```
const sub = (a : bigint, b: bigint): bigint => {
    ??
}
```

we are left to figure out how to do this…
and convince ourselves it satisfies the spec

Example: Declarative Specification (sqrt)

- Square root of x is number y such that $y^2 = x$
  - not all positive integers have integer square roots, so… let's round up
  - $(y - 1)^2 \leq x \leq y^2$

    smallest integer y such that $x \leq y^2$

```
const sqrt = (x: bigint): bigint => {

    ??

}
```

we are left to figure out how to do this…
and convince ourselves it satisfies the spec

Example: Declarative Specification (abs)

- Absolute value $|x|$ is an integer y such that
  - $y \geq x$
  - $y \geq -x$
  - $y = x$ or $y = -x$

```
const abs = (x: bigint): bigint => {
  if (x >= 0) {
    return x;
  } else {
    return -x;
  }
}
```

requires some thinking to make sure this code returns a number with the properties above

Example: Imperative Specification (HW3)

From HW3: Dijkstra's Algorithm

```
add a 0-step (empty) path from start to itself to active

while active is not empty:
  minPath = active.removeMin()  // shortest active path

  if minPath.end is end:
    return minPath  // shortest path from start to end!

  if minPath.end is in finished:
    continue  // longer path to minPath.end than the one we found before

  add minPath.end to finished  // just found shortest path to here!

  // add all paths that have one step added to this shortest path
  for each edge e in adjacent.get(minPath.end):
    if e.end is not in finished:
      newPath = minPath + e
      add newPath to active

return undefined  // no path from start to end :(
```

steps are described fully
(just translate to TypeScript)

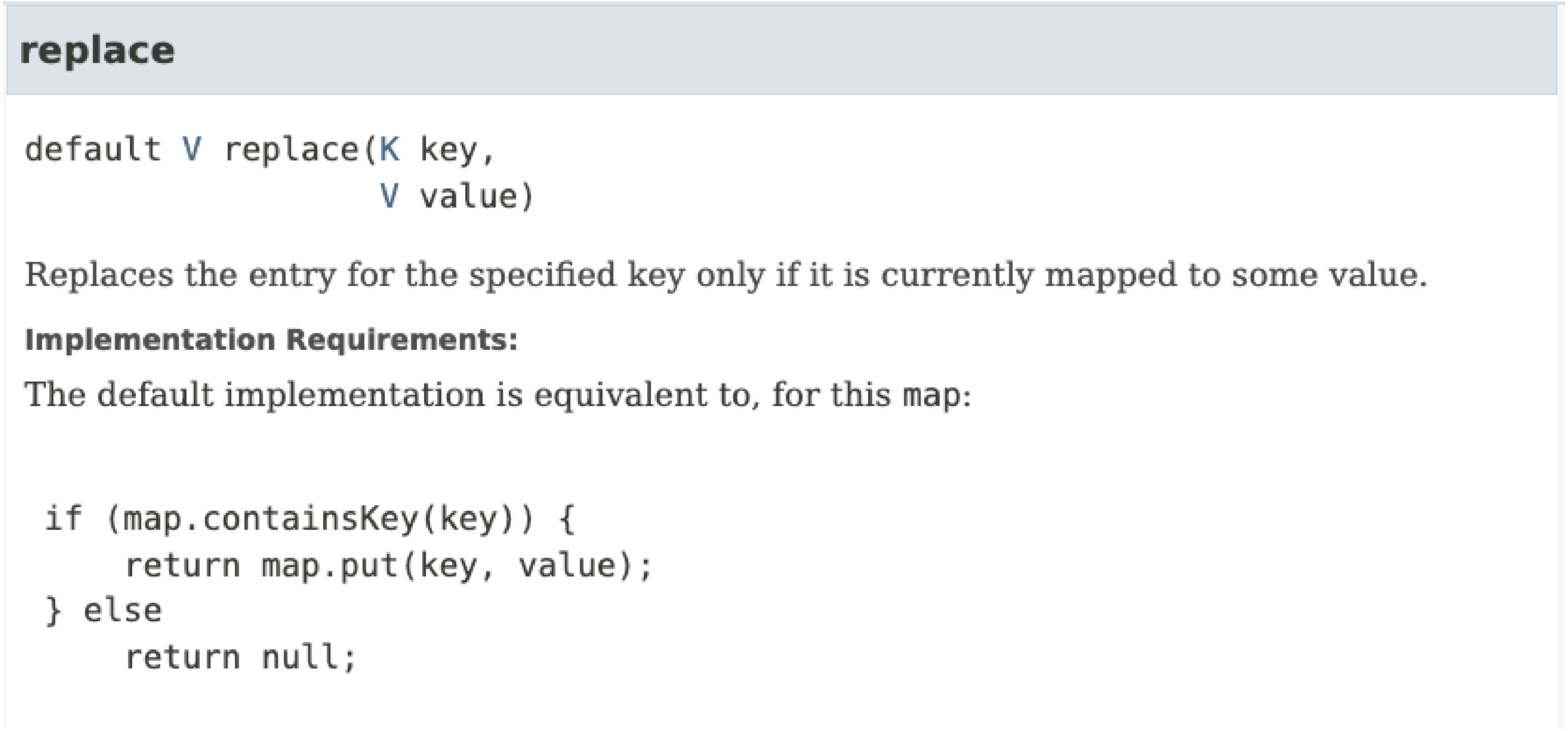
"Straight From the Spec"

- If imperative, just translate math into code
  - TypeScript here, but could also be Java
  - we often call this "straight from the spec"

- if declarative (or implementing different imperative spec), then we will need new tools for checking its correctness

Examples from Java: Map .replace()

`java.util.Map` — **set of (key, value) pairs**



replace

```
default V replace(K key,
                  V value)
```

Replaces the entry for the specified key only if it is currently mapped to some value.

Implementation Requirements:

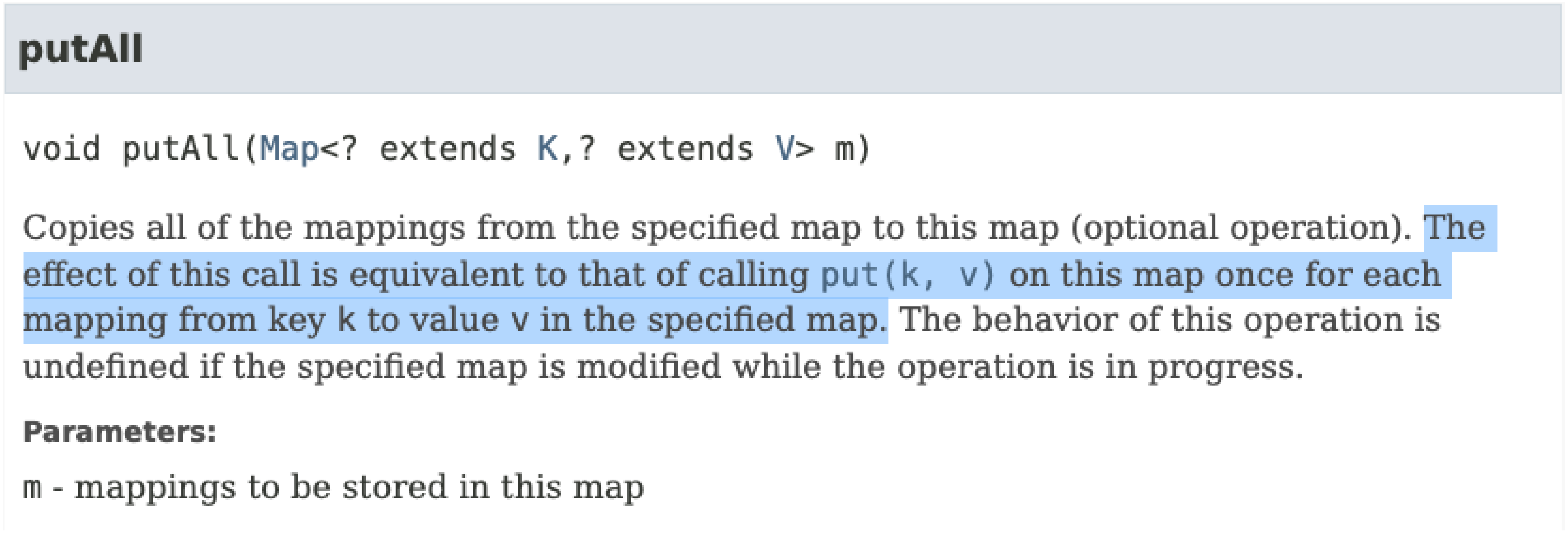
The default implementation is equivalent to, for this `map`:

```
 if (map.containsKey(key)) {
     return map.put(key, value);
 } else
     return null;
```

Imperative

Examples from Java: Map .putAll()

`java.util.Map` — **set of (key, value) pairs**



putAll

```
void putAll(Map<? extends K,? extends V> m)
```

Copies all of the mappings from the specified map to this map (optional operation). The effect of this call is equivalent to that of calling `put(k, v)` on this map once for each mapping from key k to value v in the specified map. The behavior of this operation is undefined if the specified map is modified while the operation is in progress.

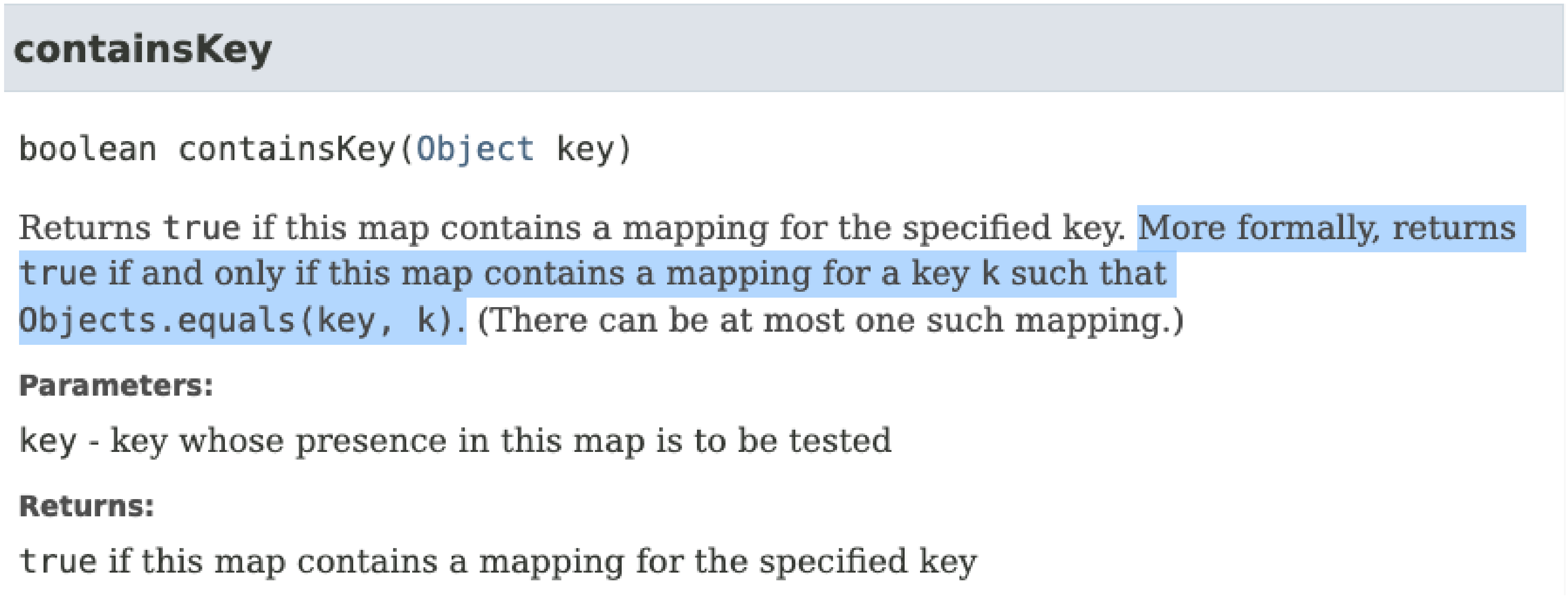
Parameters:

`m` - mappings to be stored in this map

Imperative

Examples from Java: Map .containsKey()

`java.util.Map` — **set of (key, value) pairs**



containsKey

```
boolean containsKey(Object key)
```

Returns `true` if this map contains a mapping for the specified key. More formally, returns `true` if and only if this map contains a mapping for a key k such that `Objects.equals(key, k)`. (There can be at most one such mapping.)

Parameters:

key - key whose presence in this map is to be tested

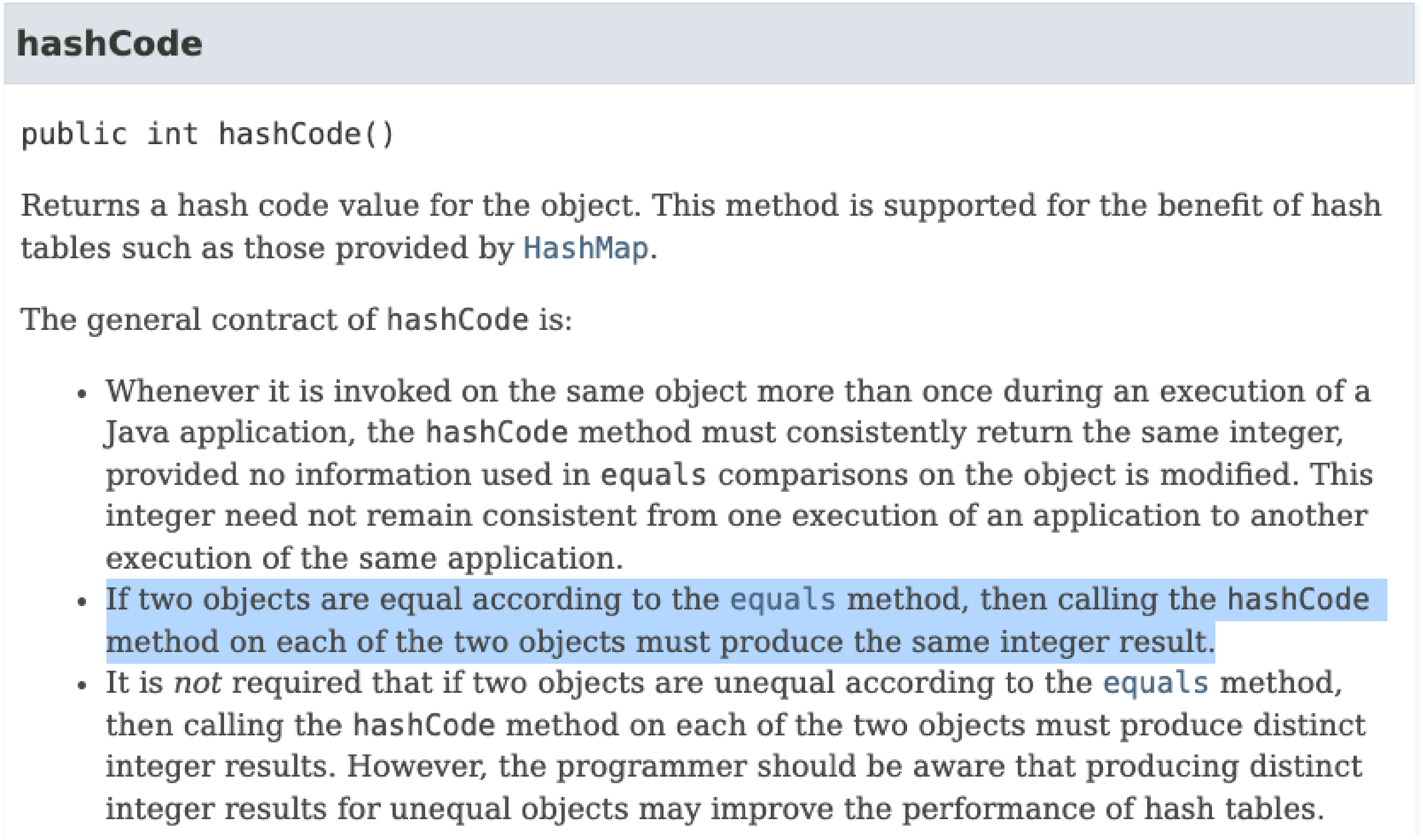
Returns:

`true` if this map contains a mapping for the specified key

Declarative (probably)

Examples from Java: Object .hashCode()

`java.util.Object`



hashCode

```
public int hashCode()
```

Returns a hash code value for the object. This method is supported for the benefit of hash tables such as those provided by `HashMap`.

The general contract of `hashCode` is:

- Whenever it is invoked on the same object more than once during an execution of a Java application, the `hashCode` method must consistently return the same integer, provided no information used in `equals` comparisons on the object is modified. This integer need not remain consistent from one execution of an application to another execution of the same application.
- If two objects are equal according to the `equals` method, then calling the `hashCode` method on each of the two objects must produce the same integer result.
- It is *not* required that if two objects are unequal according to the `equals` method, then calling the `hashCode` method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hash tables.

Declarative

This Topic's Goal: Imperative Specifications

- Toolkit for writing imperative specifications
  - define math for data and code

    write specifications that are **language independent**

    (don't want a toolkit that only works for TypeScript)
  - describe how to translate imperative specs into TypeScript

    try to make the translations as *straightforward* as possible (fewer mistakes)
  - mention new TypeScript features when related

    critical to understand what bugs the type system catches and which it does not

- Will look at declarative specifications later

Math Notation

Basic Data Types in Math

- In math, the basic data types are “sets”
  - sets are collections of objects called **elements**
  - write $x \in S$ to say that “x” is an element of set “S”, and $x \notin S$ to say that it is not.

- Examples:

$x \in \mathbb{Z}$	x is an integer	
$x \in \mathbb{N}$	x is a non-negative integer (natural)	
$x \in \mathbb{R}$	x is a real number	
$x \in \mathbb{B}$	x is T or F (boolean)	
$x \in \mathbb{S}$	x is a character	non-standard names
$x \in \mathbb{S}^*$	x is a string	

Basic Data Types in TypeScript

English	Math	TypeScript	Up to Us
integer	$x \in \mathbb{Z}$	`bigint`	
natural	$x \in \mathbb{N}$	`bigint`	non-negative
real	$x \in \mathbb{R}$	`number`	
boolean	$x \in \mathbb{B}$	`boolean`	
character	$x \in \mathbb{S}$	`string`	length 1
string	$x \in \mathbb{S}^*$	`string`	

we will often write
$x : \mathbb{Z}$ instead of $x \in \mathbb{Z}$

– **only subtraction on non-negative can produce negative**

Creating New Types in Math (Unions)

- **Union Types** $\mathbb{S}^* \cup \mathbb{N}$
  - contains every object in either (or both) of those sets
  - e.g., all strings and natural numbers

- If $\mathrm{x} \in \mathbb{N} \cup \mathbb{S}^*$, then x could be a natural or string

- Two sets can contain common elements
  - in this case, the sets are disjoint

Creating New Types in TypeScript (Unions)

- **Union Types** `string | bigint`
  - can be either one of these

- How do we work with this code?

```
const x: string | bigint = …;

// can I call isPrime(x)?
```

- We can check the type of `x` using "`typeof`"
  - TypeScript understands these expressions
  - will "narrow" the type of `x` to reflect that information

Type Narrowing With "If" Statements

- **Union Types** `string | bigint`
  - can be either one of these

- How do we work with this code?

```
const x: string | bigint = …;

if (typeof x === "bigint") {
  console.log(isPrime(x))    // okay! x is a bigint
} else {
  …                          // x is a string
}
```

Type Narrowing vs Casting

```
const x: string | bigint = …;

if (typeof x === "bigint") {
  console.log(isPrime(x))    // okay! x is a bigint
} else {
  …                          // x is a string
}
```

- Note that this does not require a **type cast**
  - TypeScript knows x is a `bigint` inside the "if" (narrowing)
- 331: there are **no type casts** (won't even show syntax)
  - unlike Java, TypeScript casts are unchecked at runtime
  - seem designed to create extremely painful debugging

Type Narrowing Gotcha

```
const f = (x: bigint): string | bigint => …;

if (typeof f(x) === "bigint") {
  console.log(isPrime(f(x)))   // why not allowed?
}
```

- TypeScript will (properly) reject this
  - no guarantee that f(x) returns the same value both times!

Type Narrowing of Function Calls

```
const f = (x: bigint): string | bigint => …;

const y = f(x);
if (typeof y === "bigint") {
  console.log(isPrime(y))       // this works now
}
```

- TypeScript can see that the two values are the same
- Functions that return different values for the same inputs are confusing!
  - maybe better to avoid that

Record Types in Math

- **Record Types** $\{\mathrm{x} : \mathbb{N},\ \mathrm{y} : \mathbb{N}\}$
  - record with fields “x” and “y” each containing a number
  - e.g., $\{\mathrm{x}\!: 3, \mathrm{y}\!: 5\}$

- Note that $\{\mathrm{x}\!: 3, \mathrm{y}\!: 5\} = \{\mathrm{y}\!: 5, \mathrm{x}\!: 3\}$ in math
  - field names matter, not order
  - note that these are not "$===$" in JavaScript

    in math, “=“ means same values

    in JavaScript, "===" is reference equality

Record Types in TypeScript

- **Record Types** `{x: bigint, y: bigint}`
  - anything with *at least* fields "x" and "y"

- Retrieve a part by name:

```
const t: {x: bigint, y: bigint} = … ;
console.log(t.x);
```

Optional Fields in TypeScript

- Records can have optional fields

```
type T = {x: bigint, y?: bigint};

const t: T = {x: 1n};
```

  - type of “ `t.y` ” is “ `bigint | undefined` ”

- Functions can have optional arguments

```
const f = (a: bigint, b?: bigint): bigint => {
  console.log(b);
};
```

  - type of “ `b` ” is “ `bigint | undefined` ”

Tuple Types in Math

- **Record Types** $\{x : \mathbb{N},\ y : \mathbb{N}\}$
  - record with fields “x” and “y” each containing a number
  - e.g., $\{x: 3, y: 5\}$

- Tuple Types $\mathbb{N} \times \mathbb{N}$
  - pair of two natural numbers, e.g., $(5, 7)$
  - can do tuples of 3, 4, or more elements also

- Mostly equivalent alternatives
  - both let us put parts together into a larger object
  - record distinguishes parts by name
  - tuple distinguishes parts by order

Retrieving Part of a Tuple

- To refer to tuple parts, we must give them names

- Tuple Types $\mathbb{N} \times \mathbb{N}$

Let $(a, b) := t$.

":=" means a definition

Suppose we know that $t = (5, 7)$

Then, we have $a = 5$ and $b = 7$

- Tuple Types `[bigint, bigint]`

```
const t: [bigint, bigint] = …;
const [a, b] = t;
console.log(a);  // first part of t
```

Simple Functions in Math

- **Simplest function definitions are single expressions**

- **Will write them in math like this:**

$$\text{double} : \mathbb{N} \to \mathbb{N}$$

$$\text{double}(\text{n}) := 2\text{n}$$

  - **first line declares the type of double function**

    takes a natural number input to a natural number output

  - **second line shows the calculation**

    know that "n" is a natural number from the *first* line

  - **will often put the type in the text before the definition, e.g.,**

    The function $\text{double} : \mathbb{N} \to \mathbb{N}$ is defined by…

$$\text{double}(\text{n}) := 2\text{n}$$

Simple Functions in Math (and shorthands)

- Another example:

$$\text{dist} : \{x: \mathbb{R}, y: \mathbb{R}\} \to \mathbb{R}$$

$$\text{dist}(p) := (p.x^2 + p.y^2)^{1/2}$$

  - first line tells us that "p" is a record and "$p.x$" is a real number

- Can define short-hand for types in math also

$$\textbf{type}\ \text{Point} := \{x: \mathbb{R}, y: \mathbb{R}\}$$

$$\text{dist} : \text{Point} \to \mathbb{R}$$
$$\text{dist}(p) := (p.x^2 + p.y^2)^{1/2}$$

Complex Functions in Math

- Most interesting functions are not simple expressions
  - need to use different expressions in different cases

- Can use side-conditions to split into cases

$$\begin{array}{ll} \text{abs} : \mathbb{R} \to \mathbb{R} & \\ \text{abs}(x) := x & \text{if } x \geq 0 \\ \text{abs}(x) := -x & \text{if } x < 0 \end{array}$$

  - conditions must be exclusive and exhaustive

    we do not want to require on *order* to determine which applies
  - there is a better way to do this in many cases…

Pattern Matching

- Can also define functions by "pattern matching"

$$
\begin{aligned}
&\mathrm{double} : \mathbb{N} \to \mathbb{N} \\
&\mathrm{double}(0) && := 0 \\
&\mathrm{double}(\mathrm{n}{+}1) && := \mathrm{double}(\mathrm{n}) + 2
\end{aligned}
$$

  - first case matches only 0
  - second case matches numbers 1 more than some $\mathrm{n} : \mathbb{N}$ …

    double(6) = double(5+1) so it matches with n = 5

    since n $\geq$ 0, we have n+1 $\geq$ 1, so it matches 1, 2, 3, …
  - pattern "$\mathrm{n}{+}2$" would match $2, 3, 4, \ldots$

- Simplifies the math in multiple ways…

Pattern Matching on Natural Numbers

- Pattern matching definition

```
double(0)    := 0
double(n+1)  := double(n) + 2
```

is simpler than using side conditions

```
double(n)    := 0                   if n = 0
double(n)    := double(n-1) + 2     if n > 0
```

  - e.g., need to explain why double(n-1) is legal

    easy in this case, but it gets harder

- We will prefer pattern matching **whenever possible**

Pattern Matching on Booleans

- Booleans have only two legal values: T and F

- Can pattern match just by listing the values:
  - the function $\mathrm{not} : \mathbb{B} \to \mathbb{B}$ is defined as follows:

$$\mathrm{not}(\mathrm{T}) := \mathrm{F}$$
$$\mathrm{not}(\mathrm{F}) := \mathrm{T}$$

  - negates a boolean value
  - no simpler way to define this function!

Pattern Matching on Records

- **Can pattern match on individual fields of a record**

$$\textbf{type}\ \ \text{Steps} := \{\text{n} : \mathbb{N}, \text{fwd} : \mathbb{B}\}$$

$$\begin{aligned}&\text{change} : \text{Steps} \to \mathbb{Z}\\ &\text{change}(\{\text{n: m, fwd: T}\}) := \text{m}\\ &\text{change}(\{\text{n: m, fwd: F}\}) := -\text{m}\end{aligned}$$

  - **clear that the rules are exclusive and exhaustive**

- **Can match on multiple parameters**
  - **e.g.**, $\text{change}(\{\text{n: m+5, fwd: T}\}) := 2\text{m}$
  - **just make sure the rules are exclusive and exhaustive**

Pattern Matching in TypeScript

- TypeScript does not provide pattern matching
  - some other languages do! (see 341)

- We must translate into "`if`"s on our own

```
type Steps = {n: number, fwd: boolean};

const change = (s: Steps) => {
  if (s.fwd) {
    return s.n;
  } else {
    return -s.n;
  }
};
```

still straight from the spec
but easy to make mistakes

Pattern Matching in TypeScript: Gotcha

$$\begin{aligned}\text{double}(0) &:= 0 \\ \text{double}(n+1) &:= \text{double}(n) + 2\end{aligned}$$

- Also need to be careful with natural numbers

```
// m is non-negative
const double = (m: bigint) => {
  if (m === 0n) {
    return 0n;
  } else {
    return double(m - 1n) + 2n;
  }
};
```

spec says double(n)
but code says double(m – 1)

  – pattern matching uses "n+1" but the code uses "m" (or "n")

    sadly, TypeScript will not let "n+1" be the argument value

Code Without Mutation

- Saw all types of code without mutation:
  - straight-line code
  - conditionals
  - recursion
- This is all that there is!
  - can write anything computable with just these
- Saw TypeScript syntax for these already…

Code Without Mutation Example

Example function with all three types

```
// n must be a non-negative integer
const f = (m: bigint): bigint => {
  if (m === 0n) {
    return 1n;
  } else {
    const n = m - 1n;
    return 2n * f(n);
  }
};
```

What does this compute?

$f(m) = 2^m$

$f : \mathbb{N} \to \mathbb{N}$

$$\begin{aligned} f(0) &:= 1 \\ f(n+1) &:= 2 \cdot f(n) \end{aligned}$$